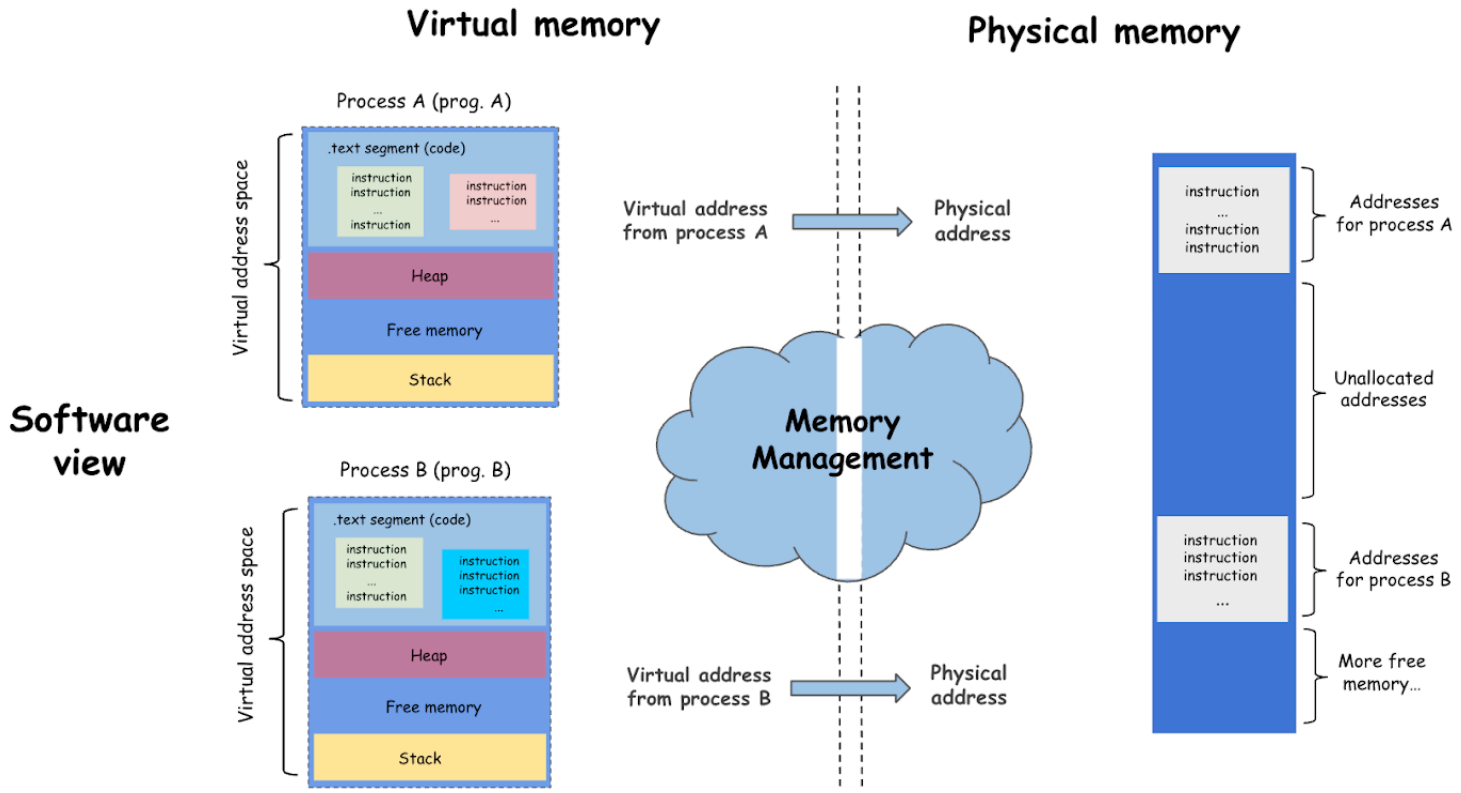
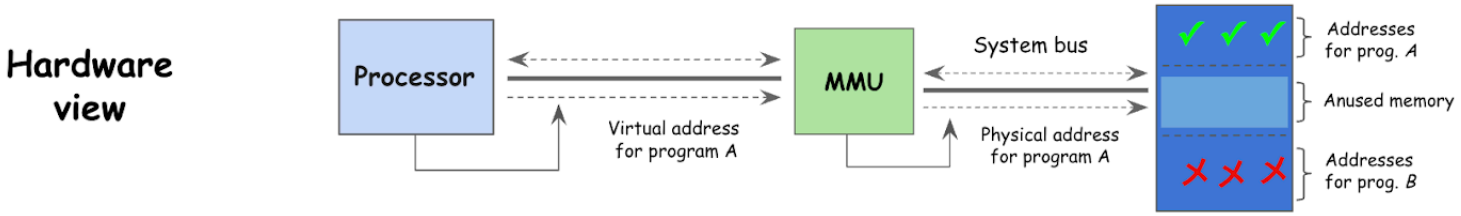
- **Q1: What is virtual memory?**
  - Virtual memory is a layer of abstraction that translates virtual addresses into physical addresses.
    - Virtual addresses is the language via which programms and processes talk to the processor about memory.
    - Physical addresses is the language via which the actual contents of memory (i.e., RAM DIMMs) are accessed by the processsor for read or write via the memory bus
  - This translation is managed by the OS and is performed with the aid of a specific hardware component, namely the Memory Management Unit (MMU).
  - The OS kernel has the privileges to turn the MMU on and off: when the MMU is on, any address produced by the processor is virtual and passes through the MMU first, before being put on the memory bus, in order to be translated into the corresponding physical memory address. By contrast, when the MMU is off, all addresses are directly treated as physical.





- In most modern commercial OSes, user programs can only talk virtual addresses to the processor, because the MMU is only off during early system booting; and user programs can only execute after the booting process is completed and the MMU is on.
- What this really means is that the way programmers imagine their programs will interact with memory is, in fact, just an illusion built on top of a transparent layer of indirection.
- For example, the following program run twice, stores a value to the variable "i" seemingly at the same memory address, but the contents are different because each process gets its own mapping of virtual to physical memory addresses.

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>

int main(int argc, char **argv) {
  int i;
  pid_t pid = getpid();

  if (argc != 2) {
    fprintf(stderr, "Usage: <%s> <argv[1] goes here>\n", *argv);
    return -1;
  }

  i = atoi(argv[1]);

  printf("[PID: %d]; &i: %p; i: %d\n", pid, &i, i)
  printf("[PID: %d]; Going to sleep for %d seconds\n", pid, i);
  sleep(i);
  printf("[PID: %d]; I am done now\n", pid, i);
  return 0;
}
```

```
➜ git:(master) ✗ echo 0 | sudo tee
/proc/sys/kernel/randomize_va_space
0

➜ vm git:(master) ✗ ./hello_vm 10 &
[1] 186394
[PID: 186394]; &i: 0xffffffffff210; i: 10
[PID: 186394]: Going to sleep for 10 seconds

➜ vm git:(master) ✗ ./hello_vm 3 &
[2] 186432
[PID: 186432]; &i: 0xffffffffff210; i: 3
[PID: 186432]: Going to sleep for 3 seconds

➜ vm git:(master) ✗ [PID: 186432]; I am done now
[2]  + 186432 done       ./hello_vm 3

➜ vm git:(master) ✗ [PID: 186394]; I am done now
[1]  + 186394 done       ./hello_vm 10

➜ vm git:(master) ✗
```

Same address / different value, both programs running...

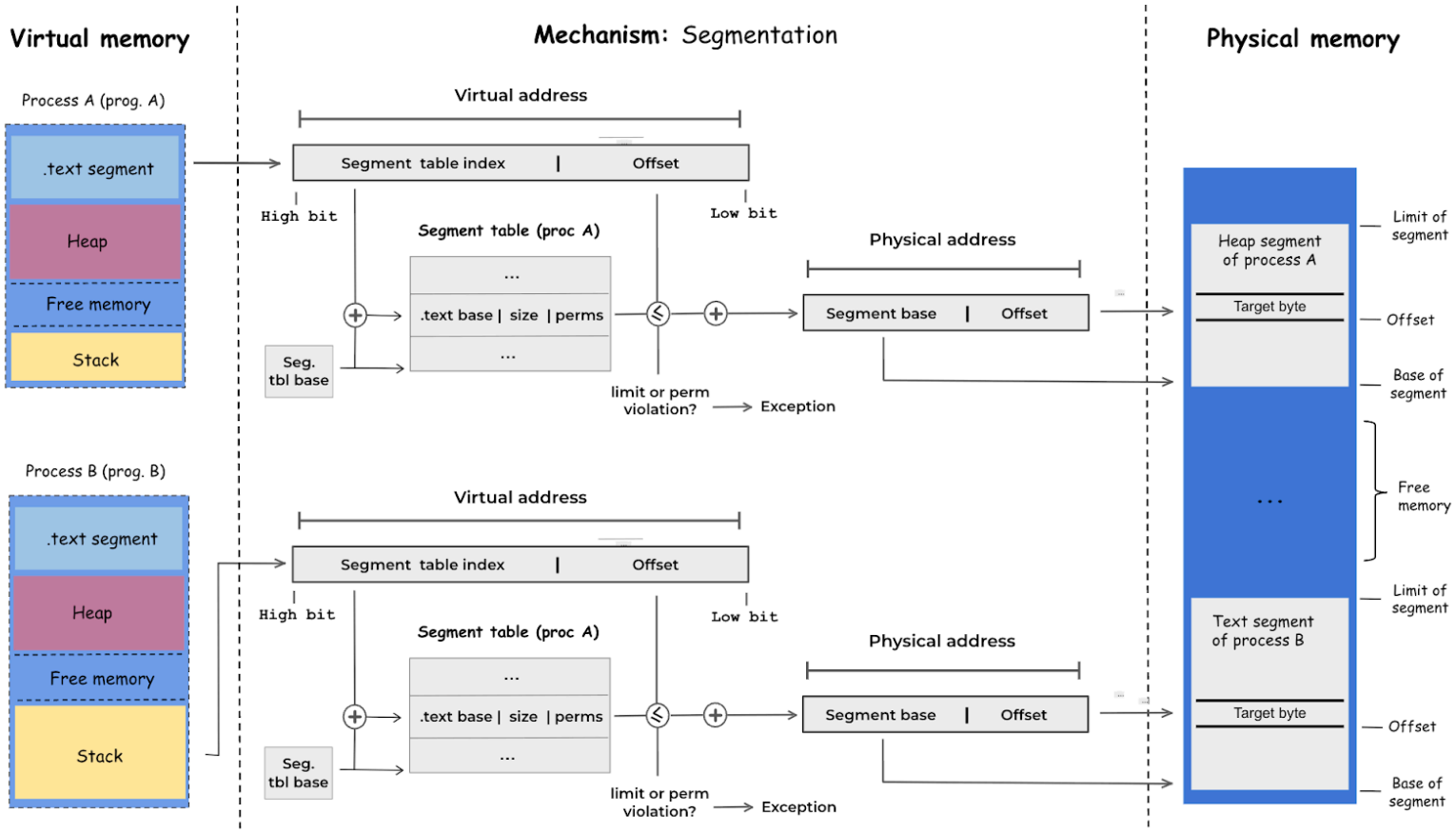
- Q2: Why is virtual memory necessary?

- **Fault Isolation**. Confinement and separation of concerns between processes (and users), such that anything that goes wrong in the address space of one process does not affect any other process present in the system.
- **Illusion of continuous memory**. Programs are written, compiled, assembled, linked, and loaded as if they have/are going to have access to abundant continuous memory; even though, in reality, physical memory is limited in size and is allocated on demand in fragmented physical blocks.
- **Frugal use of resources**. Reduce the memory footprint of large programs by avoiding allocating a physical memory range, until necessary (demand-paging), and make read-only parts of common code, that is shared across many processes (e.g., shared libraries), available with only one physical replica until a write occurs (Copy-On-Write).
- **Performant use of resources**. Leverage temporal locality code property (addresses that have been recently referenced are likely to be references again soon) and spatial locality code property (addresses that are close to the regime of recently referenced addresses are likely to be referenced in the near future) to enable programs to run almost at the speed of main memory, or even faster, at the speed of CPU caches, while regularly accessing slower, but of larger capacity, persistent storage devices (such as HDDs and SSDs) to offload parts of code that are not in imminent use. (CoW is particularly important for POSIX-compliant OSes because it significantly speeds up the starting of new processes via fork, by duplicating address spaces rapidly without actually copying the complete physical memory ranges until necessary.)

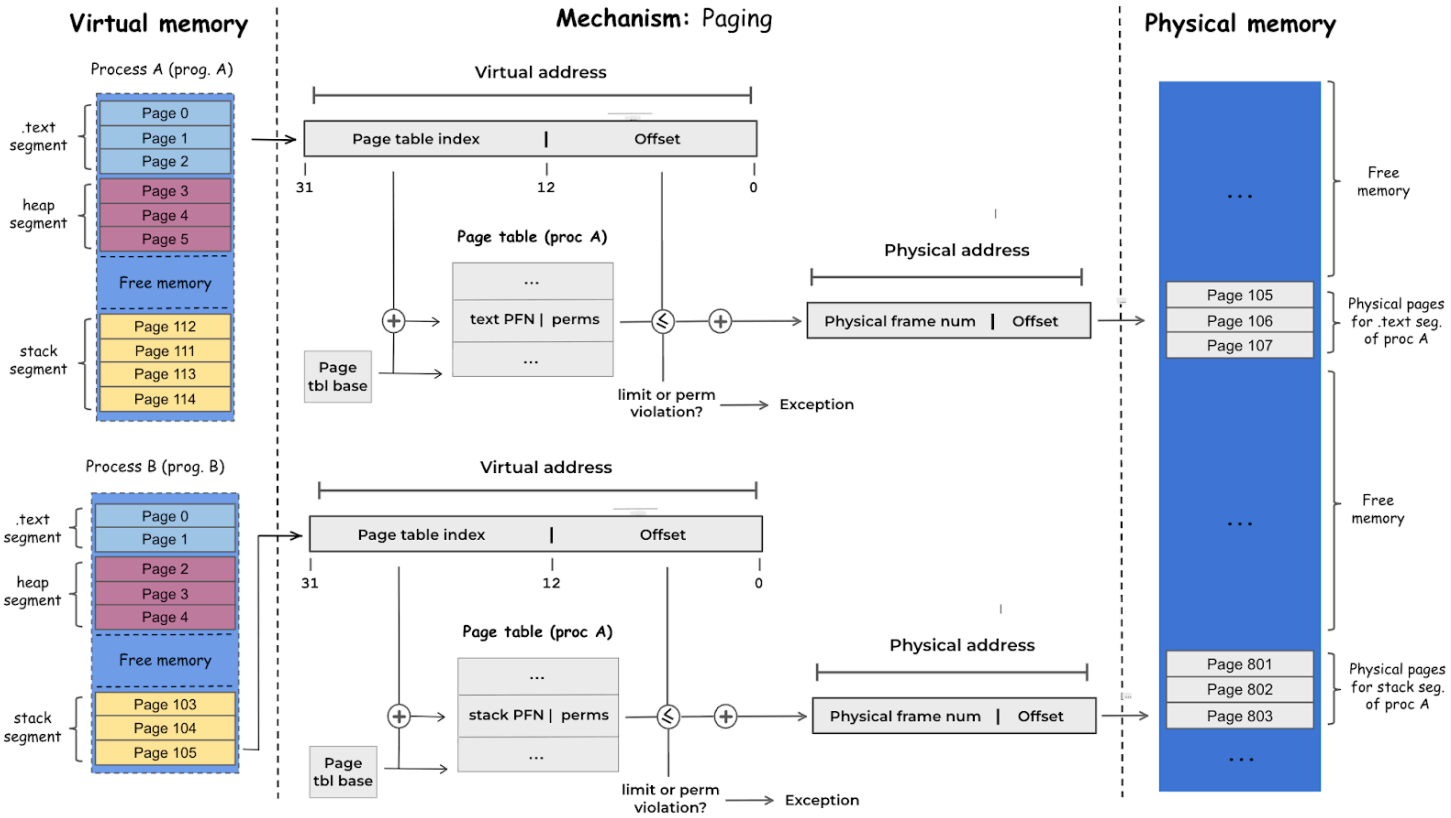
- Q3: How is virtual memory implemented?

- Although virtual memory management is conceptually simple and everything revolves around an address translation index that maps virtual to physical addresses, its implementation is quite complex.
- In fact, it is likely the most complex of all kernel subsystems, as it requires intimate cooperation between architecture-dependent software and hardware: "We've rewritten the VM several times in the last ten years, and I expect it will be changed several more times in the next few years. Within five years, we'll almost certainly have to make the current three-level page tables be four levels, etc." —Linus Torvald, 2001.
- We will use the following questions to help navigate the basic design and implementation decisions regarding the memory management kernel subsystem.
    - **A.) What are the entries of the address translation index?** Depending on the virtual memory management mechanism implemented, the OS maintains in each process's PCB either a segmentation table (segmentation), or a page table (paging), or a multilevel page table (multilevel paging), and so on.
    - **B.) How are the entries of the address translation index used?** Every time the processor needs to read or write data from or to a main memory address as a result of executing an instruction, the virtual address produced by the processor is first passed to the MMU, which, in turn, needs to look up the translation index in order to put the respective physical address on the system bus. Since address translations are extremely often and the overhead of a single main memory reference (~100 processor cycles) to look up the translation index from main memory on every address translation is impractical, a small portion of the index is cached closer to the processor in a special hardware component, called the Translation Look-aside Buffer (TLB). The TLB has a very low access time (~1 cycle) and helps so that most translations occur inside the CPU core, without the MMU having to look up the in-main-memory translation index.
    - **C.1.) How are the entries of the address translation index allocated?** New entries are added to the translation index on demand whenever a translation is needed. Specifically, the MMU first checks the TLB, and if a valid translation exists, it directly produces the respective physical address. Otherwise, if the translation is not present in the TLB or the TLB entry is invalidated [Q: when will this happen?], the hardware automatically looks up the in-memory translation index and updates the TLB. However, if the translation is either not present or invalid in the in-memory index as well, then the hardware raises an exception (i.e., a page fault) to inform the OS that it needs to allocate a new physical memory range and add the respective translation in the index.
    - **C.2.) When are the entries of the address translation index replaced?** Although main memory is quite large in size these days, there could still be cases when the available physical memory may be fully occupied, and in order for a new physical range to be allocated an eviction (replacement) of a used memory range needs to happen. In such low-memory situations, the replacement decision regarding which in-use physical memory range to evicted in order to make the necessary space available leverages the "converse" of temporal locality property: among all indexed and occupied physical memory ranges, the Last Recently Used (LRU) ones are the least likely to be referenced again soon; therefore, they are the best candidates to be evicted and swapped out (backed up to secondary storage) to make fresh space available for allocation.

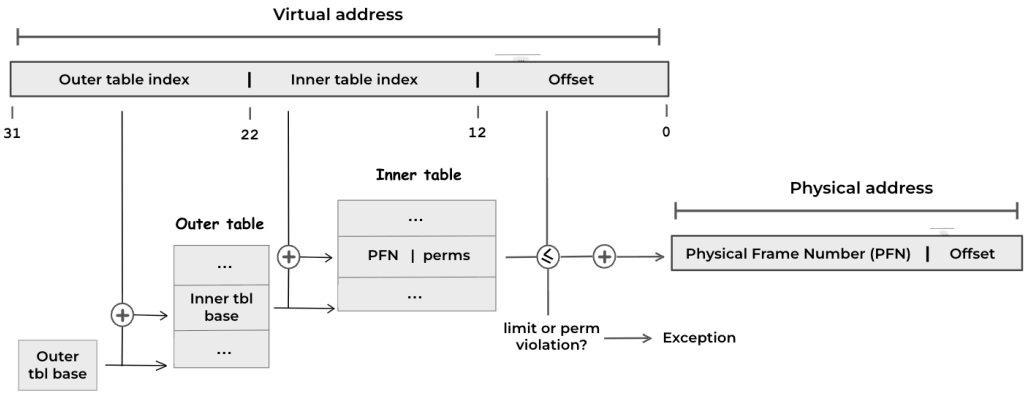
- **Q4: What are the entries of the address translation index?**
  - Obviously, keeping a tuple <virtual address → physical address> for every address translation required for every process in the system is impractical. Therefore, different ways to compress the required index size (e.g., by batching together groups of "nearby" addresses) have been invented and used over the years.
  - **Segmentation**: For each process in the system, there is a dedicated segmentation table, which holds the base and limit physical address as well as the respective permissions for each segment of the process.
    - Each time an address translation is needed, the higher bits of the virtual address (segment table index) are used to index the process's segmentation table and the respective entries of the base and size of the physical addresses as well as its permissions are retrieved.
    - The lower bits of virtual address (offset) are, then, checked against the size of the physical memory range, and the respective permissions are also asserted. If the offset is less than or equal to the segment size and no permission violation occurs, then the offset bits are appended to the segment bits, and the MMU puts the derived physical memory address on the system bus. Otherwise, a hardware exception is raised informing the OS that it needs to take action.
    - Segmentation allows each process's address space to be split into several independent chunks of different sizes and permissions, allowing for some sharing and deduplication.
    - Its main drawback is fragmentation: internal—when segments have mapped but unused parts internally; and external—multiple available segments which are unusable due to unfit sizes.



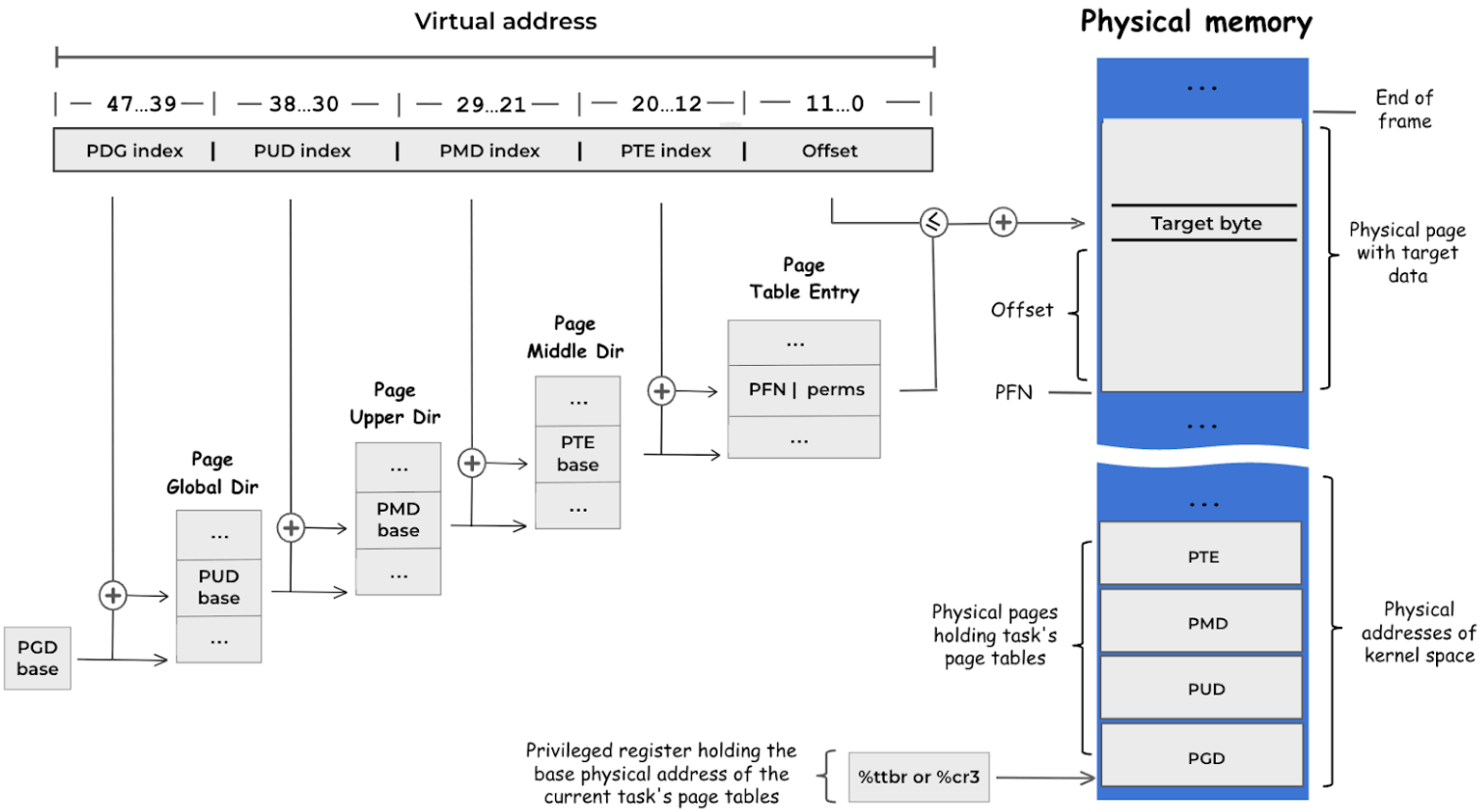
- **Paging:** For each process in the system, there is a dedicated page table, which holds the base of the respective physical address as well as the respective permissions. Unlike segmentation, where allocations of arbitrary-sized physical ranges are allowed, in paging, the physical and virtual memory are divided into fixed-sized chunks—called pages—and each segment can span multiple contiguous pages.
    - Each time an address translation needs to occur, the high bits of virtual address (page table index) are used to index the process's page table, and the respective entry of the physical page base as well as its permissions are retrieved.
    - The lower bits of the virtual address (offset) are, then, checked against the designated page size and the respective permissions are also asserted. If the offset is less than or equal to the page size and no permission violation occurs, the offset bits are appended to the lower bits of the page table entry (physical frame numbers), and the MMU puts the derived physical memory address on the system bus; otherwise, a hardware exception is raised informing the OS that it needs to take action.
    - Example: Assume that the page table entries are 4 bytes each (20 high-bits for the base of the physical page and the remaining for permissions) and that the 2nd entry in the page table of proc A is 0x0006A007. What is the translation of the 32-bit virtual addresses 0x00001402 produced by proc A? Since 0x00001400 = $[0000\ 0000\ 0000\ 0000\ 0001]_{20}$ (index=1) and $[0100\ 0000\ 0000]_{12}$ (offset), and the 2nd page table entry is 0x0006A007 = $[0000\ 0000\ 0000\ 0110\ 1010]_{20}$ (pfn = 106), then the physical translation is $[0000\ 0000\ 0000\ 0110\ 1010]_{20}$ + $[0100\ 0000\ 0000]_{12}$ = 0x0006A400.
    - Because all pages are of the same (relatively small) size, external fragmentation is not an issue with paging, and although internal fragmentation may still be an issue, it is not as prominent.
    - The main limitation of single-level paging is the size of page tables for all processes in the system: Assuming 32-bit virtual addresses, with 12 bits for the offset (4KB pages), and the 20 remaining bits for the page table index with 4-byte page table entries ⇒ for 2^20 page table entries, the OS needs to preallocate 4MB for the page table of each process in the system.



- **Multilevel paging:** Example of a two-level page table scheme with 4-byte page table entries, and 32-bit virtual addresses, where the 12 lower bits are for the offset (4 KB page size), the 10 middle bits are for the inner index, and the 10 higher bits are for the outer index bits. In this scheme, the outer table holds 2^10 = 1024 entries (4 KB total size), and each instance of the allocated inner page table also holds 2^10 entries (4 KB size). However, with a two-level paging scheme, the OS needs to preallocate one page for the outer table and an additional inner entry, on demand, for each additional 4 MB of actively used physical memory.



- **The 4-level Linux paging model:** Assume a processor architecture with 64-bit virtual addresses and 48-bit physical addresses. (The virtual address space is restricted to only 48 bits, supporting 256 TB of total addressable space; or it could be 52 bits, supporting 4 PB of total addressable space, to avoid the unnecessary cost of manufacture.)



- All page table directories, including the PGD, and the allocated PUD, PMD, and PTE instances, are held in physical pages that belong to the kernel's address space.
- Each page table directory holds 512 8-byte-long entries, and the specific structure of each entry as well as the semantics of each entries' bits are architecture-dependent: overall, on each entry, the higher bits are reserved or unused, the middle 20 bits comprise the pointer at the base of the next-level index or at the base of the physical page (for the PTE), and the remaining lower bits are used for permission checks.
- Given 8-byte-long entries, the PTE holds 512 entries which can address a 4KB page each (2 MB reach); the PMD holds 512 PTE entries which can address 2 MB each (1 GB reach); the PUD holds 512 PMD entries that can address 1 GB each (512 GB reach); and, finally, the PGD (top level) holds 512 PUD entries that can address 512 GB each (256 TB total addressable virtual address space).
- For details on the format of the page table entries expected by hardware, as well as on what each bit represents, see "Chapter 5 - Paging" in Intel Software Developer's Manual, the translation table entry formats in ARM's manual, and the Linux kernel's page tables definitions for x86 and aarch64.

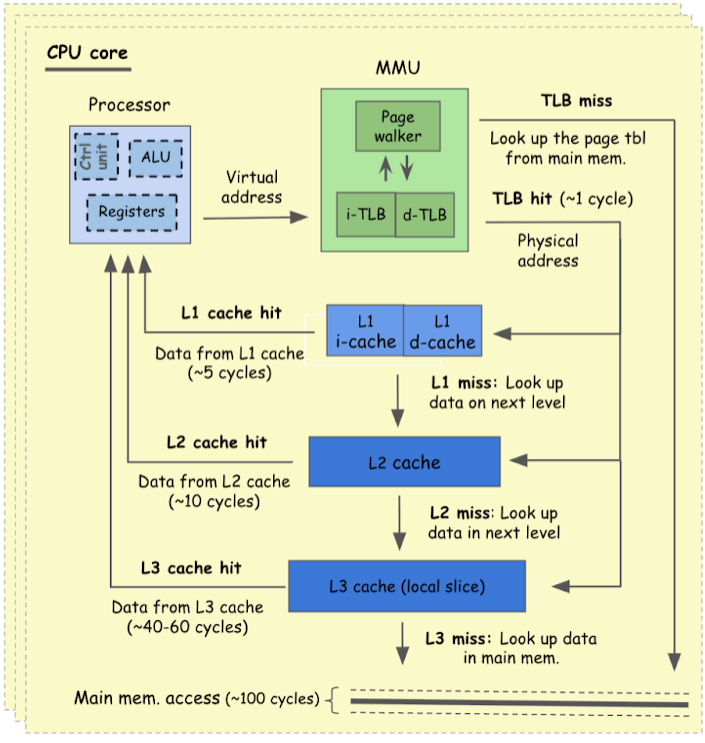
Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (see Section 5.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry (see Section 5.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 5.9.2)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 5.9.2)
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 5.8)
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 5.8)
7 (PAT)	Indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 5.9.2)
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 5.10); ignored otherwise
10:9	Ignored
11 (R)	For ordinary paging, ignored; for HLAT paging, restart (if 1, linear-address translation is restarted with ordinary paging)
(M–1):12	Physical address of the 4-KByte page referenced by this entry
51:M	Reserved (must be 0)
58:52	Ignored
62:59	Protection key; if CR4.PKE = 1 or CR4.PKS = 1, this may control the page's access rights (see Section 5.6.2); otherwise, it is ignored and not used to control access rights.
63 (XD)	If IA32_EFER.NXE = 1, execute-disable (if 1, instruction fetches are not allowed from the 4-KByte page controlled by this entry; see Section 5.6); otherwise, reserved (must be 0)

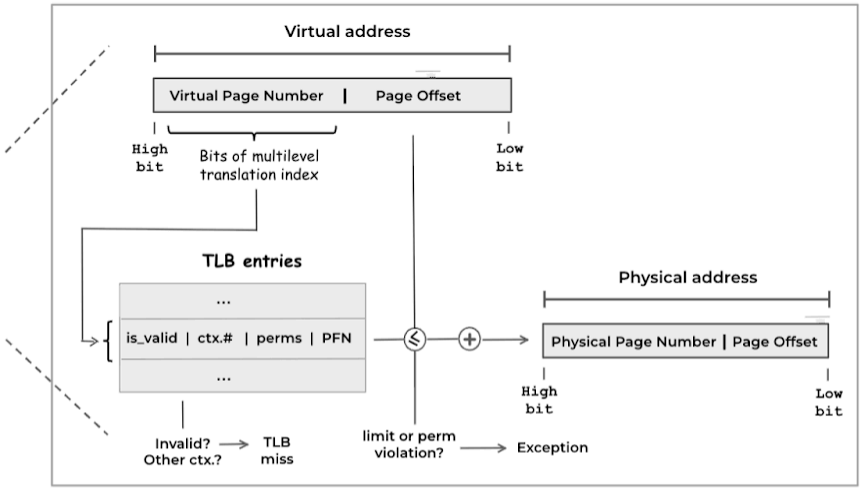
Format of a Page-Table Entry that Maps a 4-KByte Page

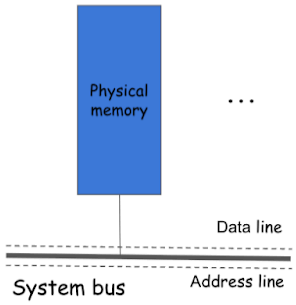
63	62–59	58–52	51–M[1]	M-1 …	… 13	12	11	10–9	8	7	6	5	4	3	2	1	0	
Reserved[2]				Address of PML4 table (4-level paging) or PML5 table (5-level paging)			Ignored						PCD	PWT	Ign.			**CR3**
XD[3]	Ignored		Rsvd.	Address of PML4 table			R[4]	Ign.		Rsvd	Ign	A	PCD	PWT	U/S	R/W	**1**	**PML5E: present**
Ignored																	**0**	**PML5E: not present**
XD	Ignored		Rsvd.	Address of page-directory-pointer table			R	Ign.		Rsvd	Ign	A	PCD	PWT	U/S	R/W	**1**	**PML4E: present**
Ignored																	**0**	**PML4E: not present**
XD	Prot. Key[5]	Ignored	Rsvd.	Address of 1GB page frame	Reserved	PAT	R	Ign.	G	**1**	D	A	PCD	PWT	U/S	R/W	**1**	**PDPTE: 1GB page**
XD	Ignored		Rsvd.	Address of page directory			R	Ign.		**0**	Ign	A	PCD	PWT	U/S	R/W	**1**	**PDPTE: page directory**
Ignored																	**0**	**PDTPE: not present**
XD	Prot. Key	Ignored	Rsvd.	Address of 2MB page frame	Reserved	PAT	R	Ign.	G	**1**	D	A	PCD	PWT	U/S	R/W	**1**	**PDE: 2MB page**
XD	Ignored		Rsvd.	Address of page table			R	Ign.		**0**	Ign	A	PCD	PWT	U/S	R/W	**1**	**PDE: page table**
Ignored																	**0**	**PDE: not present**
XD	Prot. Key	Ignored	Rsvd.	Address of 4KB page frame			R	Ign.	G	PAT	D	A	PCD	PWT	U/S	R/W	**1**	**PTE: 4KB page**
Ignored																	**0**	**PTE: not present**

Formats of CR3 and Paging-Structure Entries with 4-Level Paging and 5-Level Paging

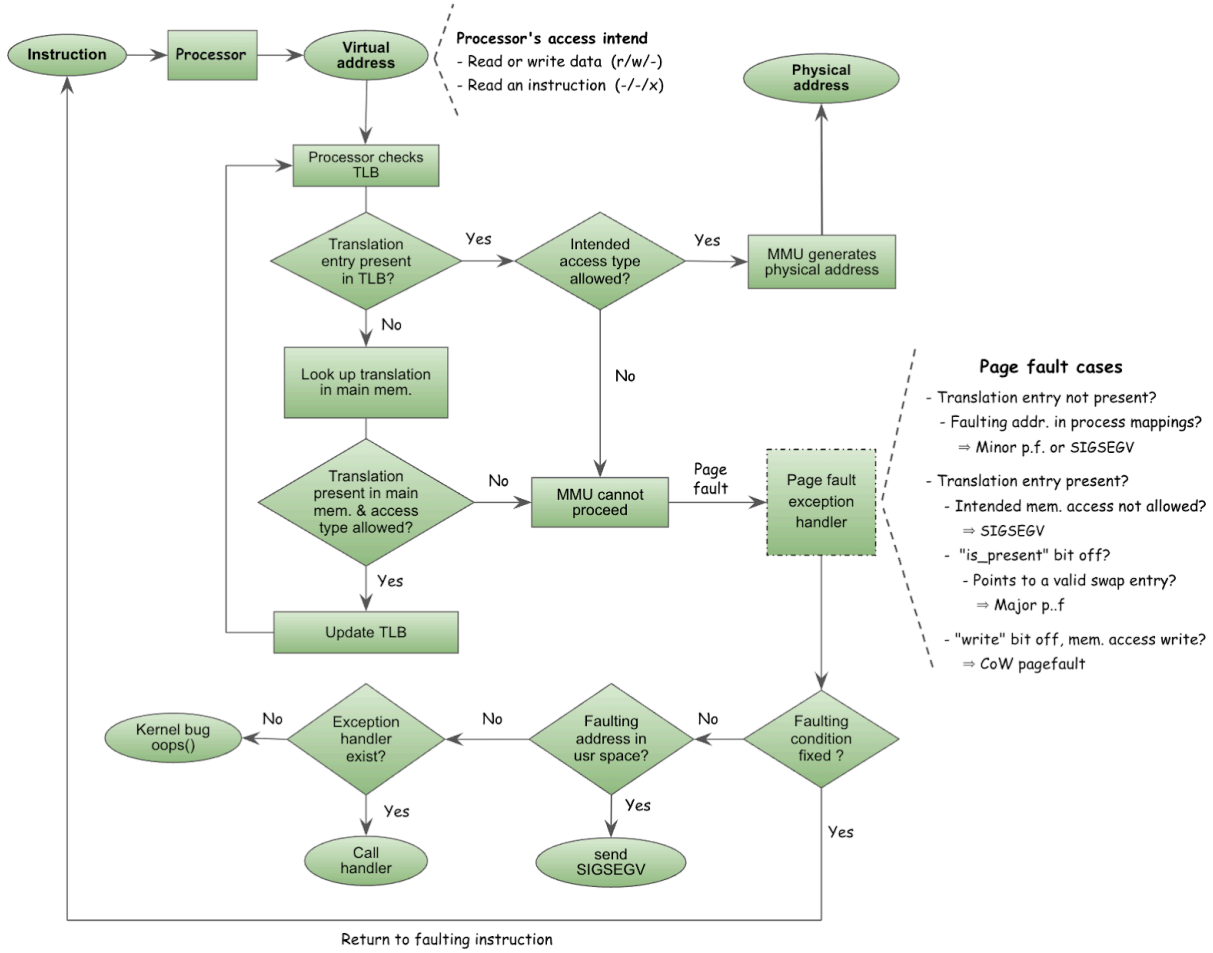
- **Q5: How are the entries of the address translation index used?**
  - Every time the processor needs to read or write data from or to a memory address as a result of executing an instruction, the virtual address produced by the processor is first passed to the MMU, which needs to look up the translation index before putting the respective physical address on the system bus.
  - Since address translations are extremely frequent and the overhead of a main memory reference (~100 processor cycles) to look up the translation index on every address translation would be impractical, a fragment of the most recent address translations is cached closer to the processor in a special hardware component, called the Translation Lookaside Buffer (TLB), which has very low access time (~1 cycle).
  - The TLB entries are of the format <valid bit, context id, permissions, physical page number>, and the TLB typically consists of a multi-level hierarchy (e.g., the 1st level is smaller and split into an i-TLB and a d-TLB, and the 2nd level is larger and caches address translations of both types).
  - Due to temporal locality, the TLB typically has a high hit rate (~99% on general workloads) and, thus, most address translations do not require the MMU to reference the translation index in main memory.
  - As long as the OS sets up the page table entries in main memory, in the case of hardware-managed TLBs, the hardware will transparently return the corresponding address translation on a TLB hit, or will walk the translation index in main memory on a TLB miss, or will raise a hardware exception on an invalid memory reference. The only responsibility of the OS is to invalidate the TLB entries every time an address space change (context switch) occurs, to prevent stale mappings of one process from spilling into the address space of another process.
  - Modern CPUs use Address Space Identifiers (ASIDs) and Process Context IDs (PCIDs) to avoid complete TLB flushes: on a context switch, the OS updates the current ASID/PCID, and the hardware ignores entries from other processes. (See this for TLB-related Linux kernel APIs.)







- **Q6: How are new entries allocated in the address translation index?**
  - New entries are allocated into the translation index lazily / on demand only when the respective translation is looked up (demand-paging); and even after a translation entry slot has been added to the index, the entry itself may remain marked "not present," if the OS can defer allocating the backing physical page.
    - Assume that a process requests 1MB for its heap—why occupy 256 physical pages before they are actually used? Just add a memory range in the allowed mappings of the process's PCB to indicate that the translation is legal, and defer the rest (including allocating a translation entry and potentially allocating the physical page backing the translation) for later.
    - Assume a process requests a 1MB zeroed-out buffer—while it only performs reads, why occupy 256 physical pages if only one zeroed-out physical page can back up all translations? Allocate all the translation entries such that they all point to the same zeroed-out physical page and mark the entry's permissions "read-only". (If a write-access occurs, then the OS will update the translation.)
    - Assume a child process is created by fork and, according to POSIX, the OS ought to duplicate the parent's address space on the child—why copying the actual physical pages from parent to the child, instead of allocating the child translation entries such that they point to the parent's physical pages, until a write access occurs on any of the two sides (Copy-on-Write). [Q: Useful? fork+exec.]
  - The above functionality (and much more) is implemented by the OS in the page fault handler, which is invoked every time a page fault exception is raised by the hardware because an appropriate translation cannot be found either in the TLB entries or in the in-memory translation index.



- To manage the translation index and implement the above functionality, the OS needs to maintain two additional data structures: One per-process data structure that keeps track of each process's allowed mappings and the respective permissions, and one global data structure that keeps track of the status of physical pages that are globally available for allocations, or not.
    - The user must be able to control what mappings (i.e., what ranges of virtual addresses) are allowed for their processes and with what permissions, at page granularity, and the OS must, in turn, add these mappings to the respective process's PCB and enforce the appropriate permission checks when each translation occurs. (The most relevant POSIX syscalls are mmap and mmprotect.)
    - The data structure that keeps track of the state of physical pages is usually managed entirely internally by the OS, and the user has no visibility into it.
- **Page fault handling.** Since every address translation may involve hardware and software collaboration, on every page fault, the exception handler needs to know (1) what the faulting (virtual) address was, and (2) what the processor's intention was when it generated the virtual address.
    - The faulting virtual address is passed by the hardware to the software via a privileged register (e.g., %cr2 in x86) whenever a page fault exception occurs.
    - The page fault handler validates that the faulting virtual address falls within a valid range that belongs to some page of a segment of the current process; otherwise, a SIGSEGV is raised.
    - The page fault handler uses the exception's error code (automatically pushed by hardware to the expedition handler's stack every time an exception occurs) to infer what the processor's intended memory access was (e.g., read or write data from an address, or read an instruction as part of its instruction fetching pipeline stage), and compares it either to the permissions registered in the respective PCB structure keeping track of the process's mappings (if no translation entry yet exists) or to the permission bits of the translation entry (if an entry is populated in the process's translation index).
    - If the translation entry is successfully updated (minor/major/CoW pagefault)—that is, the faulting condition is hopefully fixed—the "faulting" instruction is then reexecuted.

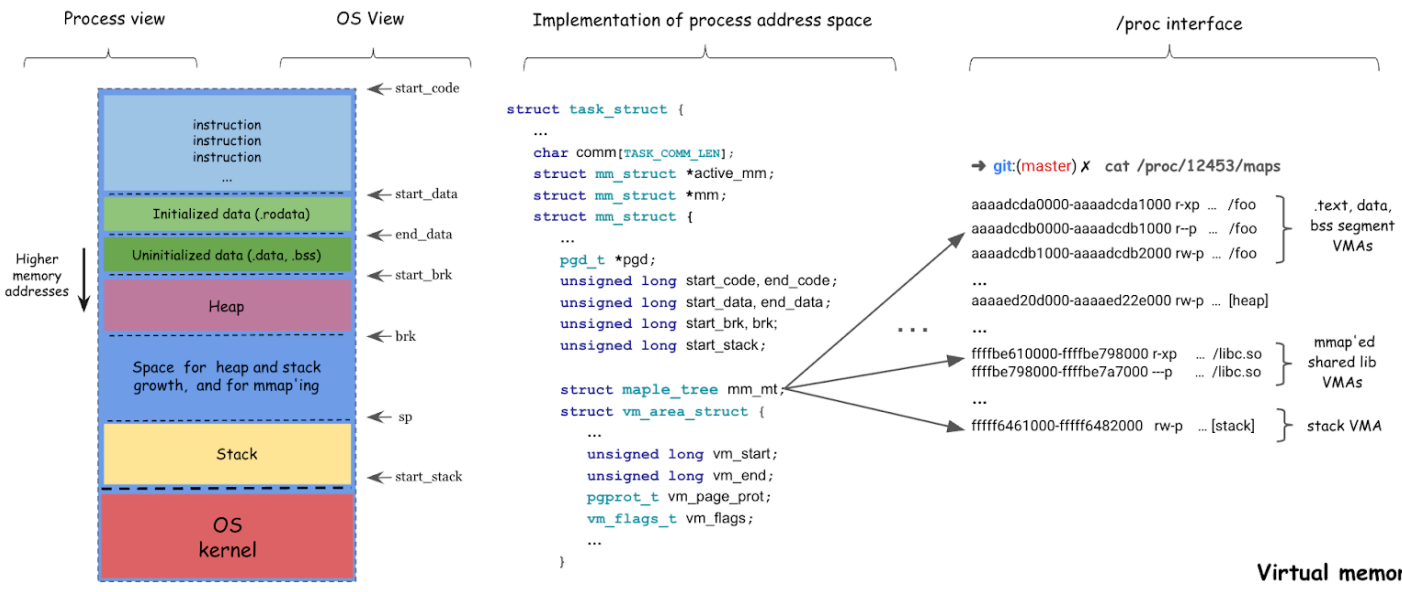
Read vs write page faults, and OS "laziness"

```c
int main(int argc, char **argv) {
 char a;
 int vma_size = 2 * 4096;
 char *buffer = mmap(NULL, vma_size,
                     PROT_READ | PROT_WRITE,
                     MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

 for (int i = 0; i < vma_size; i += 4096) {
   start_time = clock_gettime_ns();
   a = buffer[i];
   end_time = clock_gettime_ns();
   printf("page-%d: Time elapsed: %lu ns (1st read)\n",
            i / 4096, end_time - start_time);

   start_time = clock_gettime_ns();
   a = buffer[i];
   end_time = clock_gettime_ns();
   printf("page-%d: Time elapsed: %lu ns (2nd read)\n",
            i / 4096, end_time - start_time);
 }

 for (int i = 0; i < vma_size; i += 4096) {
   start_time = clock_gettime_ns();
   buffer[i] = 'A';
   end_time = clock_gettime_ns();
   printf("page-%d: Time elapsed: %lu ns (1st write)\n",
            i / 4096, end_time - start_time);

   start_time =clock_gettime_ns();
   buffer[i] = 'A';
   end_time = clock_gettime_ns();
   printf("page-%d: Time elapsed: %lu ns (2nd write)\n",
            i / 4096, end_time - start_time);
 }
}
```

```
➜ git:(master) ✗ ./read_write_page_faults

(1) minor page faults: 87, major page faults: 0

page-0: Time elapsed: 2633 nanoseconds (1st read)
page-0: Time elapsed: 78 nanoseconds (2nd read)
page-1: Time elapsed: 1956 nanoseconds (1st read)
page-1: Time elapsed: 78 nanoseconds (2nd read)        } Read pg faults

(2) minor page faults: 89, major page faults: 0

page-0: Time elapsed: 4131 nanoseconds (1st write)
page-0: Time elapsed: 113 nanoseconds (2nd write)
page-1: Time elapsed: 3694 nanoseconds (1st write)
page-1: Time elapsed: 58 nanoseconds (2nd write)      } Write pg faults
(3) minor page faults: 91, major page faults: 0

➜ git:(master) ✗ ./write_read_page_faults

(1) minor page faults: 88, major page faults: 0

page-0: Time elapsed: 5868 nanoseconds (1st write)
page-0: Time elapsed: 115 nanoseconds (2nd write)
page-1: Time elapsed: 5487 nanoseconds (1st write)
page-1: Time elapsed: 48 nanoseconds (2nd write)      } Write pg faults

(2) minor page faults: 90, major page faults: 0

page-0: Time elapsed: 90 nanoseconds (1st read)
page-0: Time elapsed: 92 nanoseconds (2nd read)
page-1: Time elapsed: 59 nanoseconds (1st read)
page-1: Time elapsed: 46 nanoseconds (2nd read)       } No read pg faults

(3) minor page faults: 90, major page faults: 0
```

- **Physical page frame eviction and replacement.** Since physical memory is finite, there could be cases where the page fault handler may need to allocate a translation entry, but no free physical page frame is available. Therefore, the OS proactively tracks physical page frame usage to be able to identify physical pages that are "good" candidates to be reclaimed in low-memory situations
    - Last Recently Used (LRU) policy**:** To identify which physical pages are "good" candidates to evict in a low-memory situation, the OS leverages an "inverse" temporal locality policy based on the premise that the Last Recently Used (LRU) page is likely the best candidate to evict.
    - Approximate LRU: Pure LRU is expensive to implement in software, and instead, an alternative approximate LRU version, known as the CLOCK algorithm, is implemented in practice with minimal hardware support.
        - Each entry in the address translation index has a "referenced" bit, which is automatically set by the MMU every time the respective translation is accessed.
        - The OS periodically sweeps through the entries of the translation index of all (or most) processes and resets the respective "accessed" bits.
        - With that, translation entries whose "accessed" bits are off will be the ones accessed less recently compared to those whose accessed bits are on, and the respective physical page frames will be preferred for reclaiming.
    - Reclaiming decision: The decision for reclaiming physical pages is made separately based on the type of occupied pages, which distinguishes between file-backed pages (i.e., data from actual file blocks) and anonymous pages without file-backed content (i.e., for heap and stack segments of resident processes).
        - File-backed pages are dropped from the kernel's page cache first, since they can be reloaded later from the corresponding file.
        - Anonymous pages that have no file backing and must be written to the swap space in order to be available for reloading are dropped next.
        - File-system-related kernel caches are dropped next. (Will discuss those later.)
        - Last resort: Out-Of-Memory (OOM) killer terminates the current process.
- **Thrashing.** Even despite all the optimization discussed so far, there could be cases when processes require more memory than the system has available for them, and thus, each time one page is brought in, another page, whose contents will soon be referenced, is thrown out.
    - When trashing occurs, processes spend all of their time blocked waiting for pages to be fetched from disk, I/O devices are at 100% utilization, but the system does no useful work.
    - Reasons for thrashing: accesses without temporal locality (past ≠ future), or cases where each process's hot memory footprint only individually fits in the system.
    - What we ordered? Memory at the size of disk with access at the speed of physical memory.
    - What did we get? Memory with the access time of the disk.

- **Q8: Memory management implementation in Linux?**
  - The address space of each process in Linux contains multiple segments, and each segment is split in smaller groups of contiguous pages represented by vm_area structs (or VMAs).
    - All the VMAs of an address space are kept track of in the mm_struct of tasks under a maple tree data structure, and multiple tasks belonging to the same address space (i.e., to the same process) will point to the same mm_struct.
    - The key property of the maple tree is that it is a range-based, compact, and cache-efficient data structure designed for fast lookups and updates while supporting lockless reads.
    - The most relevant syscalls for managing proc mappings are mmap, munmap, and mprotect.
  - Physical memory is divided into "zones" of physical pages, managed by the buddy system allocator.
    - The buddy system allocator is a range-based, power-of-two allocator: Memory is split into blocks of 2^k page size, and when a request comes in, the allocator finds the smallest block that fits; if only a larger block is available, it recursively splits it into "buddies" until the right size is reached. Conversely, on freeing, buddies are merged back into larger blocks.
    - Each zone has its own buddy allocator instance to manage free physical pages, and the core of the zoned physical page frame allocation is __alloc_pages, which takes as input a GPF flag indicating which zone physical pages must be allocated from.
  - Finally, the page table address translation index is defined here, and all generic (architecture independent) parts of the memory management is implemented here.



- The chain of calls involved in hangling a page fault exception in Linux x86 is as follows
  idt_data early_pf_idts[]
    INTG(X86_TRAP_PF, asm_exc_page_fault)
      exc_page_fault()
        handle_page_fault()
          do_user_addr_fault()
            handle_mm_fault()
              __handle_mm_fault()
                handle_pte_fault()
                  do_fault()